Uma Abordagem Reflexiva para a Construção de Frameworks para Interfaces Homem-Computador

Marília G. Coelho Cecília M. F. Rubira Luiz E. Buzato

Instituto de Computação
UNICAMP
Caixa Postal 6176
13083-970 Campinas SP
Fax: (019) 788-8442
e-mail: (mcoelho, cmrubira, buzato)@dcc.unicamp.br

Sumário

Este trabalho propõe o uso de reflexão computacional para a construção de *frameworks* orientados a objetos para interfaces homem-computador. A principal limitação desses *frameworks* é sua complexidade de projeto, que dificulta o programador da aplicação entender os mecanismos implementados no *framework*. O objetivo deste trabalho é mostrar como reflexão computacional pode ser utilizada para implementar a independência de diálogo em *frameworks* para interfaces homem-computador no meta-nível, a fim de reduzir a complexidade dos mesmos. Os serviços de visualização da interface gráfica podem ser fornecidos de forma não intrusiva e transparente através de reflexão computacional para o programador da aplicação. Este trabalho descreve inicialmente uma aplicação desenvolvida a partir do ET++, um *framework* para aplicações gráficas. Em seguida, essa mesma aplicação é descrita utilizando uma abordagem reflexiva. Uma comparação entre essas duas soluções mostra os benefícios da abordagem reflexiva.
Palavras-chave: ***frameworks* para interfaces gráficas, reflexão computacional, modelo MVC, *framework* ET++.**

Abstract

This work proposes a reflexive approach to construct object-oriented GUI application frameworks. The main limitation of these frameworks is its design complexity, that dificults its understanding by the application programmers. The goal of this work is to show how computational reflection can be used to implement the dialog independence in the meta-level for GUI frameworks, in order to reduce their complexity. The grafical interface can be provided in a transparent and non-intrusive way by means of computational reflection for application programmers. This work describes an application developed using ET++, a GUI application framework, and this same application developed using a reflexive approach. A comparison of these two is shown, and the advantages of using computational reflection is highlighted.
Keywords: GUI frameworks, computational reflection, MVC model, ET++ framework.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

1 Introdução

A necessidade de reutilização de software é uma realidade. As bibliotecas de classes provêem reutilização de componentes bem definidos e testados possibilitanto a reutilização de código, entretanto, o uso dessas bibliotecas permitem baixa reutilização de projeto. Visando ampliar o potencial de reutilização, surgiu o conceito de *frameworks* orientados a objetos. *Frameworks* orientados a objetos, segundo Firesmith [9], são coleções de classes colaborativas que capturam padrões e mecanismos que implementam os requisitos comuns de projeto em um domínio específico de aplicação. O principal obstáculo para o uso de frameworks é a chamada curva de aprendizado. Para que as decisões de projeto embutidas no *framework* sejam reutilizadas, é necessário que se saiba como o mesmo trabalha. Quanto menos específico for o *framework*, maior deve ser o conhecimento sobre como as classes colaboram entre si [14]. Dessa forma, ao se projetar um *framework* genérico, deve-se considerar a seguinte afirmação de Booch [2]: "Um *framework*, por "mais elegante" que seja, nunca será utilizado, a menos que o custo do seu entendimento e o uso das suas abstrações sejam menores do que o custo do programador construir sua aplicação do início." Em particular, no desenvolvimento de aplicações interativas, o uso de bibliotecas gráficas isoladas será escolhido, se o esforço de entender o fluxo de controle do *framework* for muito grande. Uma aplicação interativa é composta de um núcleo funcional (ou aplicação) e da interface do usuário (*GUI- Grafical User Interface*) [13]. A complexidade dos *frameworks* para construção de aplicações interativas[1] (ou *frameworks* para interfaces homem-computador) se encontra no inter-relacionamento entre as classes que implementam a funcionalidade da aplicação e as classes que implementam a funcionalidade interface gráfica. Na prática, existem muitas dependências entre esses módulos, contradizendo o conceito de independência de diálogo introduzido por Hartson e Hix [12], que diz que o software da aplicação deve ser desenvolvido independentemente do software da interface. Essas dependências dificultam e, em muitos casos, impedem que a aplicação ou interface sejam reutilizadas independentemente. A adição de uma interface gráfica a uma aplicação já implementada, por exemplo, torna-se uma tarefa extremamente trabalhosa, pois muitas alterações deverão ser feitas nos códigos fonte da aplicação. À nível de modelagem, existem modelos para o desenvolvimento de interfaces que provêem modularidade e independência de diálogo. O MVC (*Model- View - Controller*)[11], por exemplo, distribui o gerenciamento da interface gráfica sobre três componentes como mostra a figura 1. O componente Modelo denota qualquer objeto da aplicação; o componente Visão contém os objetos gráficos pertencentes à interface; e o componente Controlador controla a interação com o usuário. O componente origem de uma seta deve conhecer o componente destino da seta. Neste modelo, existem três módulos bem definidos, entretanto, na prática, as técnicas de programação atualmente utilizadas não permitem implementar esses módulos de forma que eles sejam reutilizáveis independentemente sem grande esforço.

O objetivo deste trabalho é mostrar como reflexão computacional pode ser utilizada para implementar a independência de diálogo em *frameworks* para interfaces homem-computador no meta-nível, a fim de reduzir a complexidade dos mesmos. A complexidade desses *frameworks* é reduzida quando se tem uma independência de diálogo efetiva, ou seja, quando os serviços de visualização da interface gráfica são fornecidos de forma

[1] GUI frameworks

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

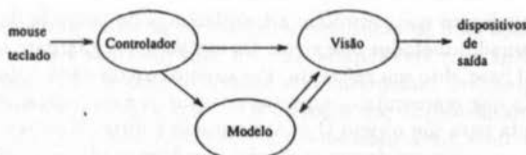

Figura 1: Interação dos componentes Modelo Visão e Controlador no MVC

não intrusiva e transparente para a aplicação. Reflexão computacional é uma técnica de programação que proporciona uma nova dimensão de modularidade no desenvolvimento de software - a separação entre o nível base e meta-nível de um software. Este trabalho compara a implementação de uma aplicação utilizando técnicas de orientação a objetos e a implementação da mesma aplicação utilizando reflexão computacional. Em ambos os casos, foi utilizado o ET++, um *framework* para construção de interfaces gráficas. Na abordagem reflexiva, foi necessária uma reestruturação da arquitetura do ET++ utilizando-se reflexão computacional para implementar o protocolo de comunicação entre a interface gráfica e a aplicação. As principais contribuições deste trabalho são: (i) as conclusões da comparação entre as duas abordagens descritas acima e (ii) proposta de um framework reflexivo para construção de interfaces homem-computador.

O restante deste trabalho está dividido da seguinte forma. A seção 2 define reflexão computacional e *frameworks*. A seção 3 mostra a estrutura e funcionamento de uma aplicação desenvolvida utilizando-se o ET++ e a seção 4 mostra a mesma aplicação desenvolvida utilizando-se o ET++ reestruturado com reflexão computacional. A seção 5 faz uma comparação das abordagens descritas nas seções 3 e 4 e, finalmente, a seção 6 conclui o trabalho e relata os trabalhos futuros a serem realizados.

2 Reflexão Computacional, Frameworks e Padrões de Projeto

Reflexão Computacional

Reflexão computacional [17, 8] é definida como sendo a habilidade de observar e manipular o comportamento computacional de um sistema através de um processo chamado de *materialização*[2]. Reflexão é baseada na noção de definir um interpretador de uma linguagem em termos dela própria. No paradigma de objetos, isso significa que a representação de classes, métodos, atributos e objetos é materializada por metaclasses e metaobjetos. A semântica do paradigma é capturada por metaobjetos, e a programação a nível de metaobjetos pode ser usada para estender e modificar essa semântica. Pode-se dizer que: (i) um sistema reflexivo possui uma representação dele próprio; (ii) o sistema reflexivo pode trazer modificações para ele próprio em virtude de suas próprias computações. Isso é feito da seguinte forma: interrompe-se o processo de execução do sistema no nível base, realiza-se computações em um meta-nível (meta-computações) e retorna-se ao nível base traduzindo os impactos dessas computações. Os elementos do meta-nível são chamados

[2]Em inglês "reification".

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

meta-objetos e os métodos que permitem a transferência do controle de execução para o meta-nível são chamados métodos reflexivos. Os metaobjetos guardam informações sobre um objeto do nível base, dito seu *referente*. Por exemplo, para cada objeto O pode existir um metaobjeto M_O que representa os aspectos estruturais e comportamentais de O. Cada mensagem s enviada para um objeto O é interceptada e dirigida ao seu metaobjeto M_O. O metaobjeto M_O encarrega-se de sua execução, que é realizada no nível base controlada pelo meta-nível. O objeto emissor da mensagem s não toma conhecimento da computação reflexiva: ele envia a mensagem solicitando serviços de um objeto e recebe o resultado esperado, sem saber que a mensagem foi desviada para um metaobjeto.

A facilidade de programar em um meta-nível permitiu o surgimento de arquiteturas que visam proporcionar transparência para o programador de aplicações na resolução de problemas como tolerância a falhas [15, 16], distribuição [6, 20, 21] e sonorização de aplicações [4]. Estes são exemplos de trabalhos relevantes que utilizam o conceito de reflexão computacional para a implementação de serviços administrativos em aplicações de software. O *framework* para sonorização de aplicações [4] permite que aplicações já existentes possam ser sonorizadas sem a necessidade de modificação dos seus códigos fontes. De forma análoga, pode-se utilizar a técnica de reflexão computacional para que uma interface gráfica seja adicionada a uma aplicação sem a necessidade de mudanças em seu código fonte. Em geral, Reflexão computacional pode proporcionar transparência na utilização dos serviços fornecidos pelo *framework*.

Frameworks e Padrões de Projeto

Um conceito importante para a compreensão de *frameworks* é padrão de projeto. Segundo Cristopher Alexander [1], um padrão[3] descreve um problema, que ocorre frequentemente em algum domínio, e a solução para tal problema de forma que possa ser utilizada muitas vezes sem ter de ser elaborada novamente. No paradigma de objetos, um padrão pode ser considerado um conjunto de regras que descrevem como realizar uma determinada tarefa no processo de desenvolvimento de software [7]. A documentação desses padrões de desenvolvimento é, na verdade, uma forma de se possibilitar a reutilização de experiência de outros projetistas em diferentes contextos. Os padrões de projeto são particularmente úteis (ou efetivos) na construção e documentação de *frameworks*. O principal propósito da abordagem de padrões de projeto baseada em frameworks é descrever o projeto do *framework* e suas classes individuais sem revelar detalhes de implementação [19]. Um exemplo de padrão é o *Observer* catalogado em [10].

Na construção de uma aplicação específica utilizando-se um *framework*, a estrutura de controle global é herdada obtendo-se a reutilização de projeto em larga escala. Essa é a principal característica dos *frameworks* que os diferencia das bibliotecas de classes orientadas a objetos. Dentre as principais vantagens da utilização de *frameworks* no desenvolvimento de aplicações podemos citar [18]: (1) infra-estrutura pré-fabricada que reduz tempo e recursos gastos com codificação, rastreamentos e testes; (2) abstração de serviços: o *framework* implementa a complexidade da aplicação; e (3) redução de custos em manutenção.

Em geral, um framework fatora padrões recorrentes em algum domínio de aplicação. Os padrões que não podem ser antecipados devem ser genéricos o bastante para que pos-

[3]Do inglês *pattern*.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

sam ser facilmente adaptados às necessidades específicas de uma aplicação. Essas partes flexíveis do *framework* são chamadas **pontos adaptáveis** (*hot spots*), os quais devem ser implementadas por métodos acoplados dinamicamente. As partes não flexíveis são chamadas **pontos fixos** (*frozen spots*). Os métodos que implementam os pontos fixos são chamados *template* e os que implementam os pontos adaptáveis, métodos *hook*. O comportamento ou o inter-relacionamento entre os objetos são definidos pelos métodos *template* que invocam os métodos *hook* para adaptar o comportamento específico da aplicação.

O programador da aplicação ao utilizar um *framework* deve conhecer sua interface e os mecanismos que ele implementa. Dessa forma, *frameworks* devem ser construídos de forma a facilitar o entendimento de seu projeto e a sua extensão de maneira que suas abstrações possam ser aplicadas a diferentes contextos. Por outro lado, sob o ponto de vista do projetista do *framework*, a construção e manutenção do próprio *framework* devem ser facilitadas. A separação das funcionalidades do *framework* em *subframeworks* e a implementação das dependências entre os *subframeworks* de forma transparente e, principalmente, a utilização de padrões de projeto facilitam a construção, documentação e manutenção do *framework*. Sob o ponto de vista do usuário, essas diretivas produzem *frameworks* mais fáceis de entender e utilizar, visto que os *frameworks* terão uma estrutura modular documentada com padrões de projeto conhecidos.

3 Um exemplo de uso do framework ET++

O exemplo de uso do framework ET++ apresentado nesta seção é um relógio com duas visões (aparências): um relógio digital e um relógio analógico. O ET++ é um *framework* para desenvolvimento de aplicações gráficas que fornece "blocos de software" pré-fabricados para a construção da interface do usuário como, por exemplo, janelas, barras de rolagem, menus, etc. O ET++ foi implementado em C++ e é executado sob várias plataformas como, por exemplo, $UNIX^{TM}$ [22]. As próximas seções descrevem a arquitetura e funcionamento da aplicação relógio.

3.1 Arquitetura da aplicação

Descrever a arquitetura básica da aplicação implica em descrever parte da arquitetura do ET++. A arquitetura básica do ET++ é composta de: blocos básicos de contrução, classes de aplicação, blocos gráficos de construção e uma camada de interface do sistema. Os blocos básicos de construção definem as estruturas de dados básicas como listas, arrays, etc. As classes de aplicação são classes abstratas que definem o comportamento genérico das aplicações desenvolvidas utilizando-se o ET++. Nessas classes se encontram os *pontos adaptáveis*, ou os métodos *hooks*, que permitem a implementação de uma aplicação específica. Os blocos gráficos de construção definem os componentes gráficos e interativos, tais como menus, caixas de diálogo, barras de rolagem, etc. Além disso, eles definem um framework para se construir outros componentes a partir dos já existentes. A camada de interface do sistema provê sua própria hierarquia de classes abstratas para serviços do sistema operacional, gerenciamento de janelas e tratamento de entrada e saída para diversos dispositivos.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

O projeto da arquitetura do ET++ foi bastante influenciado pela biblioteca de classes do Smalltalk-80 que implementa o paradigma MVC. O ET++ implementa o modelo MV, uma simplificação do MVC, que separa Modelo e Visão[4] e divide o Controlador entre esses dois componentes. A figura 2 mostra a hierarquia de classes da aplicação utilizando-se a notação UML (*Unified Modeling Language*) definida por Booch et al.[3]. O componente Modelo da aplicação corresponde às classes *Manager*, *Application*, *Clock* e *ObjTime*. O componente Visão corresponde às classes *VObject*, *TextItem*, *View* e suas especializações *ClockAnalo*, *ClockDig* e *ClockView*, respectivamente. A camada de interface do sistema, não representada na figura 2, capta os eventos do usuário e os envia para a Visão que pode tratá-los ou enviá-los para o Modelo através da classe *EvtHandler*. Todas as classes são derivadas da classe *Object*, dando um comportamento homogêneo aos objetos que herdam funcionalidades básicas implementadas nessa classe. Uma dessas funcionalidades é a dependência entre objetos. O problema de dependência entre objetos foi resolvido utilizando-se o padrão de projeto *Observer* com algumas alterações. O ET++ une as interfaces das classes do objeto observado e do observador. O método *DoObserve()*[5] é um método *hook* chamado pelo método template *Send()*. O funcionamento desse padrão de projeto será descrito na próxima seção.

Para se construir uma aplicação específica, a classe *Application* deve ser especializada. Para a aplicação relógio, foi definida a classe *Clock* (figura 2) como especialização da classe *Application*, a qual funciona como *Manager* da aplicação. A aplicação *Clock* possui um relógio representado pela classe *ObjTime* do ET++. Um objeto da classe *ObjTime* possui um *Timer*, objeto responsável pelos eventos síncronos e assíncronos do sistema, a classe *Timer* e sua hierarquia não estão representados na figura, por motivos de espaço. Os métodos *hooks* redefinidos na classe *Clock* são *DoMakeContent()* e *DoMakeMenuBar()*. *DoMakeMenuBar* possui uma implementação *default* na classe *Manager*, neste caso, sua redefinição foi necessária para que nenhuma barra de menu fosse acrescentada à janela. *DoMakeContent()* cria o conteúdo visual da aplicação, um objeto da classe *View* que possui um relógio digital (*ClockDig*) e um relógio analógico (*ClockAnalo*), visões do relógio propriamente dito representado por *ObjTime*.

Observe que existem, nesta implementação, 3 dependências entre Modelo e Visão nas classes específicas da aplicação. Note também que elas são necessárias, pois a aplicação deve conhecer sua visão (relacionamento 1) e as visões do relógio precisam recuperar a hora do sistema e portanto conhecer o relógio (relacionamentos 2 e 3).

3.2 Funcionamento da Aplicação

Esta seção descreve a sequência de mensagens que implementa as principais operações da aplicação através de diagramas de mensagens de objetos definidos na notação UML [3]. Um diagrama de mensagens de objetos, é um cenário que mostra a seqüência de mensagens que implementa uma operação específica.

O primeiro método executado, *Main*, é definido no arquivo que contém as definições

[4]Em estruturas mais simples como um menu composto de vários itens de menu não existe a separação em Visão e Modelo devido à simplicidade da estrutura de dados.

[5]Por convenção, os métodos do ET++ cujo nome iniciam com "Do" são métodos hook que podem ser redefinidos

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

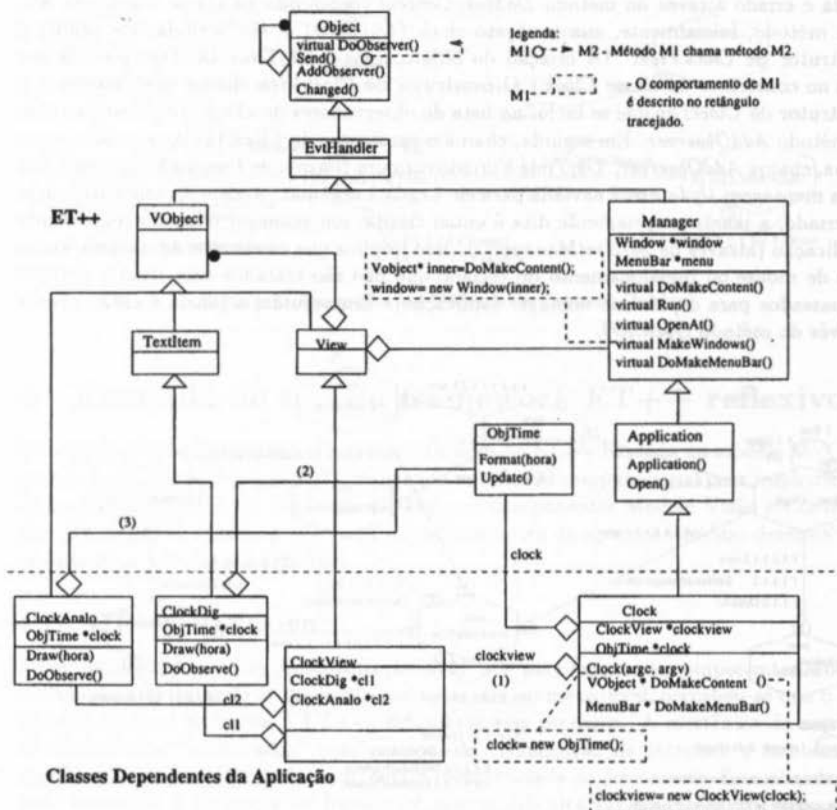

Figura 2: Diagrama parcial de classes da aplicação Relógio

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

das classes dependentes da aplicação. O único comando de Main é : return Clock(argc, argv).Run(). O construtor da classe *Clock* é então chamado, este, por sua vez, chama o construtor de *Application* que inicia o sistema operacional e de janelas. Em seguida, o método *Run* é chamado para o objeto *Clock*. A sequência de execução do método *Run()* pode ser visualizada na figura 3. Inicialmente, chama-se o método *Open()*, a partir do qual cria-se a janela da aplicação (métodos representados por $X1$). O conteúdo da janela é criado através do método *DoMakeContent* (redefinido na classe Clock) de $X1$. Esse método, inicialmente, cria o objeto clock (*ObjTime*) e, em seguida, ele chama o construtor de *ClockView*. (A criação do objeto relógio *ObjTime* também poderia ser feita no construtor da classe *Clock*.) O construtor de *ClockView* chama primeiramente o construtor de *ClockDig* que se inclui na lista de observadores do clock (*ObjTime*) através do método *AddObserver*. Em seguida, chama o construtor de *ClockAnalo*, que, da mesma forma, chama *AddObserver*. *ObjTime* é criado com um *timeout* de 1 segundo, isso significa que a mensagem *Update* será enviada para ele a cada 1 segundo. Após o conteúdo da janela ser criado, a janela propriamente dita é então criada, seu *manager* é setado como sendo a aplicação (através de *SetNextManager()*). Isso implica que os eventos do usuário, como *cliks* de mouse ou pressionamento do teclado que não são tratados pela janela, deverão ser passados para o próximo *manager* (aplicação). Em seguida, a janela é então aberta através do método *OpenAt()*.

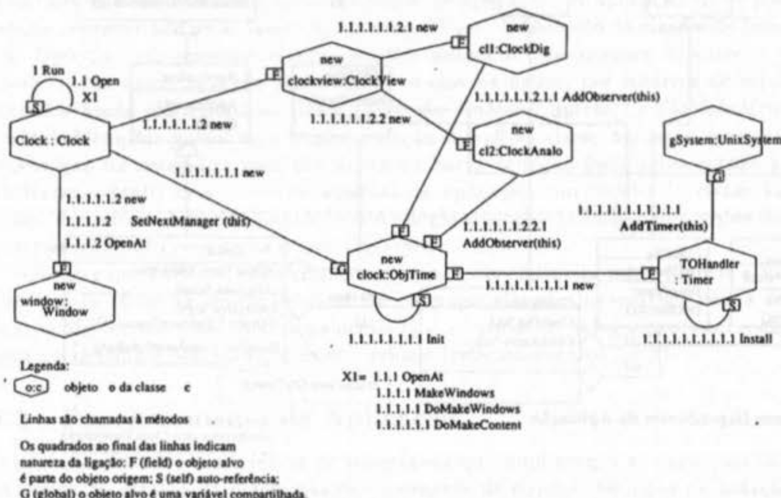

Figura 3: Diagrama de objetos do método Run() da classe Clock

Após a iniciação do sistema e construção da janela, é iniciado, através do método *Control()* (Figura 4), um laço responsável pelo tratamento dos eventos do sistema e do usuário que é finalizado quando a aplicação termina. Esse laço envia sempre a mensagem *CheckTimer* para o objeto *Timer* que, caso tenha se passado 1 segundo, envia a mensagem *Update* para *clock*. Os métodos *Changed* e *Send* do padrão *Observer* (definidos na classe

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

Object) são então chamados. *Send* chama o método *DoObserve* para cada observador na lista de *clock*. *DoObserve* recupera a hora exata do sistema, através do método *Format* e seta a *string* com a hora atualizada para o relógio digital, ou redesenha o ponteiro para o relógio analógico.

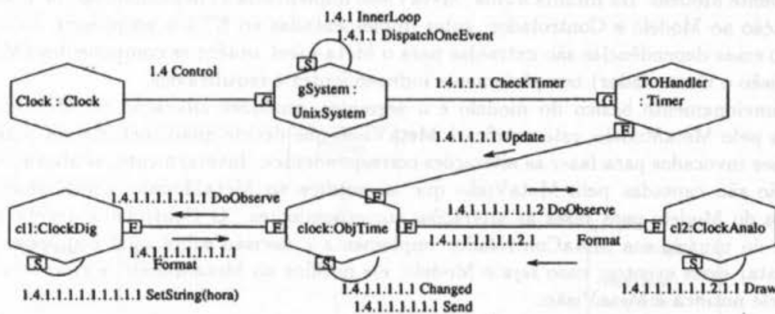

Figura 4: Diagrama de Objetos do método Control

4 Exemplo de uso do framework ET++ reflexivo

Esta seção propõe primeiramente a reestruturação do ET++ baseada no modelo MVC (figura 1) (e não no MV (seção 3.1)), utilizando-se reflexão computacional para implementar de forma transparente as dependências entre os componentes Modelo Visão e Controlador. Em seguida, mostra a estrutura e funcionamento da aplicação relógio desenvolvida utilizando-se o *framewok* reflexivo.

4.1 ET++ Reflexivo

Ao se iniciar o processo de reestruturação do ET++ para separar os componentes Modelo e Visão, implementando as dependências entre eles no meta-nível, percebeu-se que o MV (modelo no qual se baseia o ET++) dificultava esse processo. A existência de serviços do Controlador "misturados" com serviços de visualização da interface e também do modelo reduz a modularidade e aumenta a complexidade do *framework*. Essa abordagem pode aumentar a eficiência do *framework*, porém dificulta sua compreensão, extensão e manutenção. Detectadas essas limitações do modelo MV, decidimos adotar o modelo original MVC para reestruturar o ET++, visando reduzir sua complexidade e facilitar seu entendimento. A utilização de reflexão computacional, para a implementação das dependências entre os componentes, se deve ao fato dela proporcionar uma nova dimensão de modularidade – o meta-nível – permitindo que a separação dos componentes proposta pelo modelo MVC seja obtida efetivamente a nível de implementação.

A figura 5 mostra a arquitetura reflexiva do ET++ baseado no MVC. O *framework* reflexivo contém no nível base os objetos da aplicação (componente Modelo), os objetos da interface (componente Visão) e os objetos de interação com o usuário (componente

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

Controlador). Esses componentes são independentes e a comunicação entre eles é feita através dos componentes MetaModelo, MetaVisão e MetaControlador, implementados no Meta-nível. O MetaModelo implementa as dependências do Modelo em relação aos outros dois componentes (Visão e Controlador), antes implementadas no ET++ no próprio componente Modelo. Da mesma forma, MetaVisão implementa as dependências da Visão em relação ao Modelo e Controlador, antes implementadas no ET++ na própria Visão. Quando essas dependências são extraídas para o Meta-nível, obtêm-se componentes (Modelo, Visão e Controlador) completamente independentes e reutilizáveis.

O funcionamento básico do modelo é o seguinte: qualquer alteração no Modelo é captada pelo MetaModelo, este notifica à MetaVisão que decide quais métodos da Visão devem ser invocados para fazer as alterações correspondentes. Inversamente, as alterações da Visão são captadas pela MetaVisão que as notifica ao MetaModelo o qual chama métodos do Modelo para fazer as alterações correspondentes. O Controlador recebe os eventos do usuário e o MetaControlador implementa a decisão sobre qual componente deve tratar esses eventos: caso seja o Modelo, ele notifica ao MetaModelo, e caso seja a Visão, ele notifica à MetaVisão.

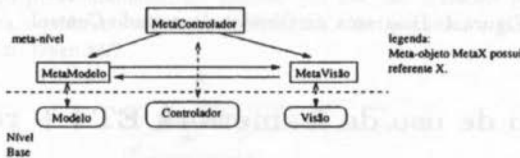

Figura 5: Arquitetura do Framework

4.2 Aplicação Relógio utilizando-se o ET++ Reflexivo

A Figura 6 mostra a implementação da aplicação relógio utilizando-se o ET++ reflexivo. Observe que todas as dependências entre Modelo e Visão foram extraídas do nível base e levadas para o meta-nível. Dentre as classes específicas da aplicação, foram eliminadas as dependências (1), (2) e (3) mostradas na figura 2. A aplicação não mais precisa conhecer sua Visão e as visões do relógio não mais precisam conhecê-lo. Dentre as classes do ET++, todas as dependências podem, da mesma forma, ser levadas para o meta-nível. Observe na figura 2 que uma aplicação (Modelo) conhece sua janela e barra de menu (Visão) e a visão (classe *View*) deve também conhecer seu *manager* (aplicação). Essas dependências são implementadas no meta-nível e a conexão com o nível base feita através dos métodos reflexivos *OpenAt* da classe *Manager* e *Update* da classe *ObjTime*. (Utilizamos a notação de OpenC++ versão 1.2 para identificar um método reflexivo.) O método *OpenAt* permite que o controle de execução seja levado para o metaobjeto que cuidará da construção da janela, barra de menus e outros objetos visuais. Da mesma forma, o método *Update* permite que o controle de execução seja levado para o meta-nível, computações sejam feitas e as alterações nas visões sejam efetuadas. Esses processos são explicados adiante.

A figura 7 mostra o projeto parcial do MetaModelo e seu relacionamento com a *MetaVisão*. A classe *MetaManager* possui uma referência para os meta-objetos *MetaWindow*

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

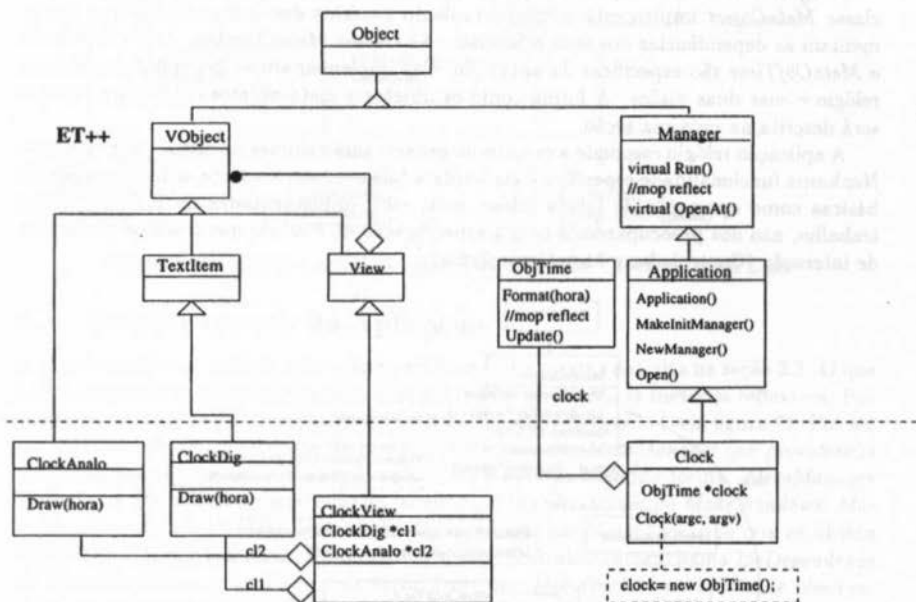

Figura 6: Diagrama parcial de classes da aplicação Clock no nível base.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

e *MetaMenubar* através dos quais envia mensagens para os objetos *Window* e *MenuBar* da interface. O método *MakeWindows* da classe *MetaManager* cria a janela da aplicação, ele é um método *template* que invoca o método *hook DoMakeContent*, o qual deve ser redefinido para construir o conteúdo da janela. A classe *MetaAplication* implementa as dependências, entre Modelo e Visão, comuns às aplicações em geral. As dependências específicas da aplicação Clock são implementadas pela classe *MetaClock* através dos métodos *DoMakeContent* e *DoMakeMenuBar*. As mensagens da aplicação (*Clock*) são passadas para a Visão (*ClockView*) através de *MetaClock* e *MetaClockView*.

A figura 8 mostra a implementação do padrão de projeto Observer no metanível. A classe *MetaObject* implementa o comportamento genérico dos meta-objetos que implementam as dependências dos seus referentes. As classes *MetaClockDig*, *MetaClockAnalo* e *MetaObjTime* são específicas da aplicação, elas implementam as dependências entre o relógio e suas duas visões. A forma como os objetos e meta-objetos colaboram entre si será descrita na próxima seção.

A aplicação relógio responde a eventos do usuário como cliques de mouse, por exemplo. Nenhuma funcionalidade específica é atribuída a tais eventos, somente as funcionalidades básicas como operações de janela (close, exit, etc), implementadas no ET++. Neste trabalho, não nos preocuparemos com a especificação do módulo que trata os periféricos de interação (Controlador e MetaControlador).

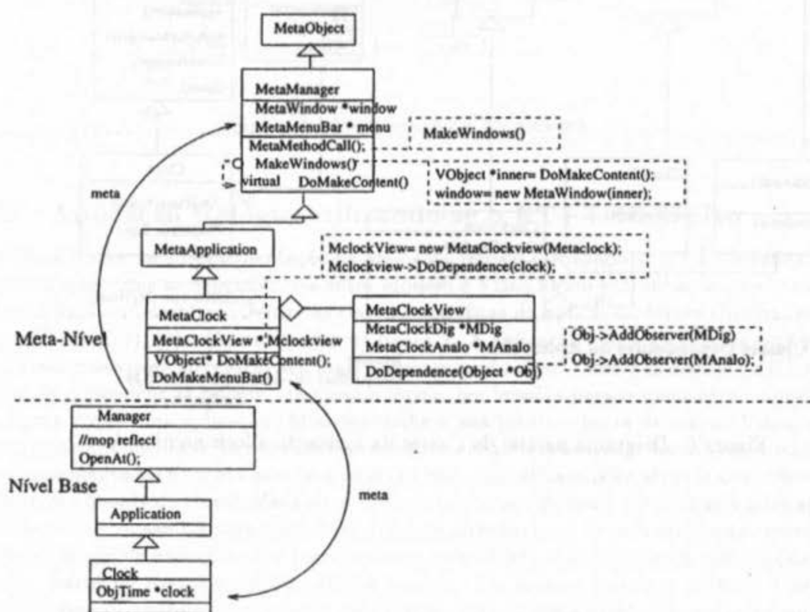

Figura 7: Diagrama parcial de classes da aplicação Clock no nível base.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

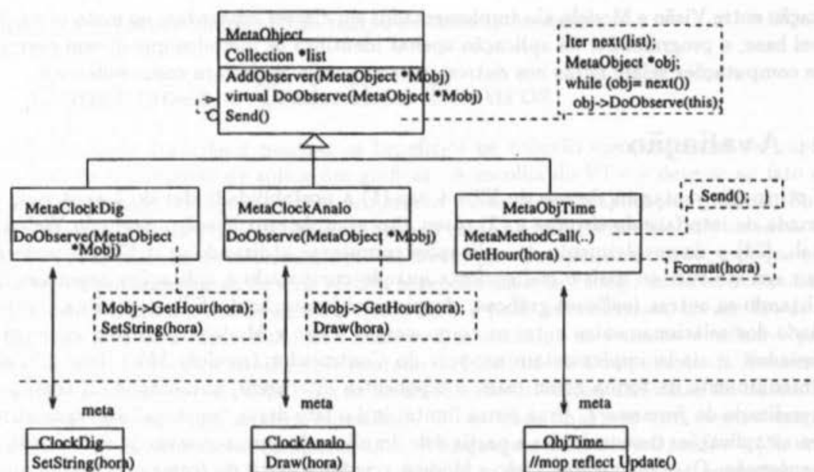

Figura 8: Diagrama parcial de classes da aplicação Clock no nível base.

4.3 Funcionamento da Aplicação

A sequência de execução da aplicação é praticamente a mesma descrita na seção 3.2. O que a diferencia são as interrupções efetuadas pelo meta-nível para os métodos reflexivos. Por exemplo, na construção da aplicação, o método *OpenAt* é invocado (veja figura 7). Por ser um método reflexivo, o controle de execução é tomado por *MetaManager* que providencia a criação dos objetos visuais da aplicação. Isso é feito da seguinte forma: *MetaManager* "solicita" à *MetaClock* a construção da janela da aplicação através de *MakeWindows*. *MakeWindows* chama *DoMakeContent* (*hook*), redefinido em *MetaClock*, que cria os objeto *ClockView* e *MetaClockView*. Após criar as visões, *DoMakeContent* chama *DoDependence* de *MetaClockView*, o qual adiciona *MetaClockDig* e *MetaClockAnalo* à lista de observadores de *clock* (*ObjTime*). Em seguida, o controle de execução volta para o nível base e o método *Control* é chamado. A cada 1 segundo, a mensagem *Update* é passada para o *clock* (*ObjTime*). A execução desse método reflexivo será interrompida por *MetaObjTime* (figura 8) que invocará o método *Send*. *Send* invoca o método *hook DoObserve* para os meta-objetos dos observadores do relógio (*MetaClockDig* e *MetaClockAnalo*). *DoObserver*, redefinido em *MetaClockDig* recupera a hora atual através de *MetaObjTime* e em seguida chama *SetString(hora)* de *ClockDig*. Semelhantemente, *DoObserver*, redefinido em *MetaClockAnalo*, recupera a hora, através de *MetaObjTime*, e chama *Draw(hora)*. Esse procedimento se repete até a finalização da aplicação.

Para o programador da aplicação, a implementação mostrada na figura 6 é mais simples que a descrita na seção anterior, mostrada na figura 2. Observe, na figura 6, que nas classes da Visão existem somente operações relativas a ela, idem para o Modelo. Diferentemente da implementação da figura 2 onde nas classes da Visão existem chamadas a métodos do Modelo. Utilizando-se o *framework* reflexivo, códigos referentes à comu-

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

nicação entre Visão e Modelo são implementados em classes separadas, no meta-nível. No nível base, o programador da aplicação apenas identifica os métodos que devem permitir que computações sejam feitas nos outros componentes e os declara como reflexivos.

5 Avaliação

As principais vantagens do uso do ET++ são (1) a portabilidade devido à existência da camada de interface do sistema e (2) o seu alto grau de reutilização. Segundo Weinand et al. [23] o desenvolvimento de aplicações complexas utilizando-se o ET++ pode reduzir em até 80% ou mais o código fonte quando comparado a aplicações desenvolvidas utilizando-se outras *toolboxes* gráficas. A principal limitação do ET++ está na complexidade dos relacionamentos entre os componentes Visão e Modelo que estão fortemente acoplados, e ainda implementam serviços do Controlador (modelo MV). Isso dificulta o entendimento da forma como esses componentes interagem, aumentando o tempo de aprendizado do *framework*. Uma outra limitação é o fato dessa "mistura" ser transmitida para as aplicações desenvolvidas a partir dele, implicando num aumento da dificuldade de manutenção. Os componentes Visão e Modelo, criados a partir do *framework*, são também fortemente acoplados de modo que eles não podem ser reutilizados independentemente, sem grandes alterações. Suponha a situação onde se deseja adicionar uma interface gráfica a uma aplicação já implementada. Para se utilizar o ET++, muitas alterações teriam de ser feitas nos códigos fontes da aplicação, pelo fato dos seus componentes não serem independentes.

A reestruturação do ET++ utilizando reflexão computacional foi proposta a fim de solucionar ou minimizar os problemas acima citados. O uso do MVC ao invés do MV modulariza o *framework*, visto que os serviços fornecidos pelo Controlador são implementados separamente dos componentes Visão e Modelo. O entendimento do *framework* é facilitado quando o código de controle (dependência entre os componentes) são implementados no meta-nível e serviços da aplicação (os próprios componentes) no nível base. A construção de aplicações complexas é facilitada, pois as tarefas de projeto e implementação podem ser divididas entre três grupos distintos de projetistas: os projetistas da interface gráfica, os projetistas da aplicação e os projetistas das dependências entre aplicação e interface gráfica. Os projetistas que implementam a interface gráfica (componente Visão) não precisam conhecer a interface da aplicação (componente Modelo) e vice-versa. Somente os projetistas que implementam as dependências entre os dois componentes (no meta-nível) precisam conhecer a interface de ambos. Um outro benefício dessa abordagem é a possibilidade da reutilização dos componentes construídos a partir do *framework* independentemente. Uma aplicação já implementada poderia, sem grande esforço, ser acoplada a uma interface gráfica. Para tanto, alguns métodos da aplicação teriam de ser declarados como reflexivos e o MetaModelo implementado, além da interface (Visão e MetaVisão). Em suma, pode-se dizer que o *framework* reflexivo proposto neste trabalho é mais fácil de entender, manter e estender. Sua limitação seria uma possível perda de eficiência caso o número de interrupções efetuadas pelo meta-nível seja muito grande. Futuramente, pretende-se estender o *framework* para o desenvolvimento de aplicações gráficas distribuídas. Neste caso, alguns experimentos práticos na literatura [6] mostram que o overhead associado às interrupções efetuadas pelo meta-nível é

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

insignificante quando comparado ao tempo de latência da rede.

6 Conclusões e Trabalhos Futuros

O objetivo deste trabalho é mostrar os benefícios de reflexão computacional na implementação de *frameworks* de aplicações gráficas. A escolha do ET++ deve-se ao fato de ele ser considerado um dos mais bem projetados *frameworks* de aplicações gráficas. Suas limitações são devidas (1) às decisões de projeto como adoção do modelo MV ao invés do MVC; e (2) às técnicas de implementação utilizadas (composição, associação, etc) que não permitem a implementação de uma efetiva independência de diálogo entre os componentes. O objetivo dos autores foi mostrar que o ET++ pode ser melhorado no sentido de se tornar mais fácil de utilizar, estender e manter. Um protótipo do *framework* reflexivo proposto neste trabalho está sendo implementado no Instituto de Computação da Unicamp. Para não construir o *framework* a partir do zero, serão utilizados conjuntos de classes do ET++ que implementam serviços específicos. O resultado final será uma reestruturação do ET++ utilizando reflexão computacional. A linguagem utilizada é OpenC++ [5], uma extensão de C++. A escolha de OpenC++ deve-se ao fato de o ET++ ser implementado em C++.

Como dito anteriormente, os trabalhos futuros incluem a extensão do *framework* para o desenvolvimento de aplicações gráficas distribuídas. Neste caso, além das dependências entre os componentes Modelo, Visão e Controlador, os serviços relativos à distribuição também poderão ser implementados no meta-nível. Quanto a eficiência do *framework* reflexivo, estudos feitos por Shiba e Masuda em [6] mostram que o *overhead* associado ao MOP (protocolo de meta-objetos) de OpenC++ é irrelevante quando ele é utilizado para computação distribuída, pois, embora não seja pequeno, é insignificante se comparado ao tempo de latência da rede.

Referências

[1] C. Alexander, S. Ishikawa, and M. Silverstein. A pattern language. Technical report, Oxford University Press, 1977.

[2] Grady Booch. Designing an application framework. *Dr. Dobb's Journal*, 19(2), February 1994.

[3] Grady Booch and James Rumbaugh. Unified Method for Object-Oriented Development - Documentation Set - Version 0.8. Technical report, Rational Software Corporation, 1995.

[4] Marcelo Campo and Roberto Tom Price. Um ambiente para sonorização não intrusiva de aplicações orientadas a objetos. In *Submetido para X Simpósio Brasileiro de Engenharia de Software*, São Carlos - Brasil, Outubro 1996.

[5] Shigeru Chiba. Open C++ programer's guide. Technical Report 93-3, Department of Infomation Science- University of Tokyo, 1993.

[6] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level arquitecture. *ECOOP'93*, July 1993.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

[7] Peter Coad. Object oriented patterns. *Comunication of the ACM*, (9), September 1992.

[8] Jacques Ferber. Computational reflection in class based object oriented languages. *OOPSLA'89 Proceedings*, October 1989.

[9] Donald G. Firesmith. Frameworks: The golden path to object nirvana. *JOOP*, (6), October 1993.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Publishing Company, April 1994.

[11] A. Goldberg. *Smalltalk-80 - The Interactive Programming Enviroment*. Addison Wesley, 1983.

[12] H. Rex Hartson and Deborah Hix. Human-computer interface development: Concepts and systems for its manegement. *ACM Computing Surveys*, 21(21):5–92, March 1989.

[13] Deborah Hix and H. Rex Hartson. *Developing User Interfaces: ensuring usability through product and process*. John Willey and Sons, 1993.

[14] Ralph E. Johnson. Documenting frameworks using patterns. 1992.

[15] Maria Lúcia Blanck Lisbôa and Cecília M. Fischer Rubira. Técnicas de orientação a objetos para tolerância a falhas. In *I Simpósio Brasileiro de Linguagens de Programação- I SBLP*. Belo Horizonte, MG, Setembro 1996.

[16] Maria Lúcia Blanck Lisbôa, Cecília M. Fischer Rubira, and Luiz E. Buzato. Arquitetura reflexiva para o desenvolvimento software tolerante a falhas. In *XXIII Seminário Integrado de Software e Hardware- SEMISH XXIII*. Recife, PE, Agosto 1996.

[17] Pattie Maes. Concepts and experiments in computacional reflection. *ACM SIGPLAN Notices, OOPSLA'87*, 22(12):147–155, December 1987.

[18] H. E. Orfali. *The Essential Distributed Objects Survival Guide*. Wiley, 1996.

[19] Wolfgang Pree. *Meta patterns - a means for capturing the essentials of reusable object-oriented design*. Springer- Verlag, 1994.

[20] Robert Stroud. Transparence and reflection in distributed systems. *Fifth ACM SIGOPS European Workshop*, April 1992.

[21] Yasuhiko Yokote Fumio Teraoka and Mario Tokoro. A reflective arquitecture for object-oriented distributed operating system. *ECOOP'89 -Proceedings of the Third European Conference on Object-Oriented Programing*, 1989.

[22] André Weinand, Erich Gama, and Rudolf Marty. An object-oriented application framework in C++. *OOPSLA'88- ACM Sigplan Notices*, (11), November 1988.

[23] André Weinand, Erich Gama, and Rudolf Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured programming*, (2), 1989.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor